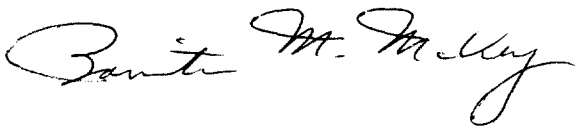
Betraying the Eye: Visual Threshold Encryption Schemes

An Honors Thesis (HONRS 499)

By

Robert P. Miller

Thesis Advisor
Dr. Bonita McVey



Ball State University
Muncie, IN

May 1999

Expected Date of Graduation: May 8, 1999

Visual Threshold Encryption Schemes – Table of Contents

1) Introduction
2) Overview of EncryptThis
3) Program Details

Appendix A – EncryptThis Examples
Appendix B – EncryptThis Source Code and Software

Introduction

Encryption is not something that is thought about by most people on a daily basis. In fact, most people never think about it; they don't have to. Most encryption and decryption takes place in such a way that the person who is taking advantage of it never even realizes it. As computers take over more and more of our everyday living, however, encryption continues to play a much larger part in every individual's life. Different types of encryption can be found in actions ranging from a student logging into his/her school's network to a child trying to decode a friend's secret message.

As technology continues to advance, encryption becomes much more important to larger scale operations such as national security. Therefore, better encryption methods are continuously being sought out in order to help protect sensitive information, regardless of who it belongs to. These new methods vary, but their goal is always the same.

This text explains one of these new methods of encryption which involves what is called a visual threshold scheme. As the term suggests, this form of encryption is used to take images and modify them in such a way that a person viewing the encrypted image cannot tell what the original image was. However, under certain circumstances specified by the person performing the encrypting, the original image can be viewed.

At this point there may be some question as to why a form of encryption specific to images is necessary when any image could be represented as data which other techniques could easily encrypt in a form which may be just as secure. The goal of this encryption technique is not to replace all other techniques used to encrypt images (in fact, the method has certain shortcomings which allow it to only be used on certain types of

images). Like so many other encryption methods, this is simply another tool at the disposal of those who wish to protect their data.

Overview

The visual threshold scheme which EncryptThis uses was first brought to the public in 1996 but is based on work in encryption reaching back to as early as 1979. In the visual threshold scheme used by EncryptThis, the encryption process takes place on a purely black and white image. By "purely" it is meant that grayscale images cannot be used. In the process of encrypting the original image, a certain number of encrypted images are produced. Because of the way the encryption scheme works, this number of encrypted images will always be greater than one. By taking certain combinations of these encrypted images and overlaying them like transparencies, a person can then see the original image. In this way, the person who encrypts the original image can distribute each of the encrypted images to a person with the knowledge that only certain combinations of these people can see the original image.

Before a person can encrypt an image using this visual threshold scheme, there are two things that must be considered: the number of encrypted images that should be produced and the combinations of these images that will be able to form the original image. After determining these, the person can encrypt the image, creating a series of black and white images. By examining any set of these images, you cannot determine anything about the original image unless that set is one of those which the person encrypting the original image decided would form the original image. This holds true for computers as well. No matter what method is used, a computer cannot determine anything about the original image when given the same restrictions mentioned previously.

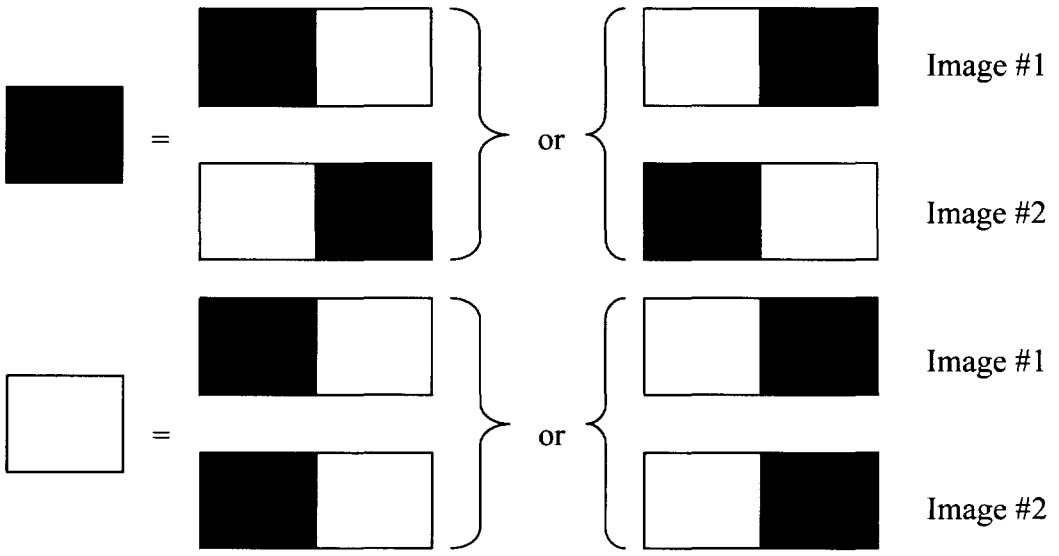
There are really two main problems with this encryption scheme. The first is that it only handles black and white images. It seems reasonable that people would like to encrypt grayscale or color images as well but this method simply will not work on them with the necessary security restrictions. The second problem is that, in the encryption process, some amount of error is introduced into the encrypted images. In other words, the image seen when overlaying qualified sets of encrypted images will not be exactly the same as the original image. This is also due to the security restrictions.

Although it has its limitations, the visual threshold scheme used by EncryptThis is very secure and less involved than other encryption schemes. The only way to determine what the original image looked like is to have a qualified set of encrypted images. Although it might seem simple to obtain a certain number of images to make up a qualified set, it is much the same as someone trying to obtain a private key of some sort in another encryption scheme. Beyond that, it is possible that it would be more difficult to obtain a qualified set of images since it is likely that each image in a qualified set would have to be taken from a different source. The encryption scheme is less involved than other schemes because, unlike most schemes, the computer itself does not do the decrypting. Instead, the power of the human visual system is used to decode the image seen and make sense of it.

Details

When an image is encrypted using EncryptThis, it is not only split into multiple images (shares) but is also expanded horizontally such that the width of each share is greater than the width of the original image. This expansion is the result of taking each pixel of the original image and dividing it into some number of subpixels.

The easiest way to see this is by example using a 2 out of 2 visual threshold scheme. In a 2 out of 2 visual threshold scheme, the original image is split into two encrypted images (this is where the second 2 comes from). Both of these images must be laid on top of each other to form the original image (this is where the first 2 comes from). In order for this to work, each pixel of the original image is divided into two subpixels. The possible subpixel combinations for a black and white image are shown below:



When a black pixel occurs in the original image, the encrypted images get one of the two possible combinations of subpixels. The same occurs when a white pixel is encountered but with different possible combinations of subpixels. Wherever a black pixel occurs in

the original image, it will occur in the image produced by overlapping the two shares. Any white pixels in the encrypted images are treated as transparent so that black pixels will be seen through them. The problem with the final image's quality surfaces when we look at the possibilities for the white pixel. When one encrypted image is laid over another, a white pixel from the original image comes out as half white and half black in the final image. This cannot be helped, however. No information about the original image can be obtained by having just a single one of these images if we have these combinations of subpixels. Setting up the subpixels in any other way, though, would allow a person to look at a single encrypted image and at least make some educated guess at what the pixel was in the original image.

This specific example can be generated using the general algorithm for producing visual threshold schemes. In order to encrypt an image using this approach, EncryptThis executes a series of steps. In general these steps are:

- Determine the qualified sets of participants

  This is done by asking the user for the sets and then converting the results to a matrix which can be easily handled by the program.

- Determine the maximal forbidden sets

  The maximal forbidden sets are those sets which contain the largest possible number of participants in them but still do not produce the original image. The program determines these sets using an algorithm involving the previously obtained qualified sets.

- Determine the cumulative array matrix

  The cumulative array matrix is used to form the final basis matrices which are used to actually encrypt the image.

- Construct the basis matrices for a k out of k visual threshold scheme

  Each visual threshold scheme has two basis matrices (one for white pixels and one for black pixels) which are used in the encryption process. There is a simple method for determining the basis matrices used in encrypting an image when all of the encrypted images are required to reproduce the original image. This method is described later.

- Construct basis matrices for current t out of k visual threshold scheme

  Constructing the basis matrices used to actually encrypt the original image involves both the cumulative array matrix and the basis matrices previously constructed for a k out of k visual threshold scheme.

- Encrypt the original image

  After constructing the basis matrices, the image can finally be encrypted.

The running time of each of these steps is highly dependent on the total number of shares being generated and marginally dependent on the number and size of the qualified sets of shares which will produce the original image.

The first step is relatively simple. The user creates different selections of shares which are each stored as an array. When it is time to encrypt the image, each of these arrays is translated into a row of ones and zeros in a qualified matrix. A 1 in a row signifies that the ith share (where i is the column holding the 1) along with all other

columns from that row having a 1 in them are a qualified set which can form the original image. A zero in a column signifies that the ith share is not a required element of that qualified set.

Determining the maximal forbidden sets is a tedious process. EncryptThis uses a method which is not entirely efficient but will find the maximal forbidden sets if they exist. This method involves generating a matrix in which each row is a permutation of some number of shares into the total number of shares to be created. The number of shares chosen for the permutation starts at the total number of shares and is decremented each iteration until the maximal forbidden sets are found. After a matrix of permutations is generated, each row is examined to see if it is forbidden. A row is forbidden if all the shares which have a 1 in their column will not generate the original image. If the current row is forbidden, then it is removed from the matrix. If, after all the rows have been examined and have been determined to be either qualified or forbidden, at least one row still exists in the matrix, then the rows from this matrix will be the maximal forbidden sets. However, if there are no rows left in the matrix, then the process is repeated after generating another matrix using the decremented value. If no maximal forbidden sets can be found, then the visual threshold scheme cannot be handled by EncryptThis.

The cumulative array matrix is one in which each row represents in which maximal forbidden sets a share does not exist. Each share has a row assigned to it and each column represents one of the maximal forbidden sets. If a certain share is not a member of one of the maximal forbidden sets, then a 1 is placed in the appropriate spot in the matrix. This matrix is easily be determined by taking the transpose of the maximal forbidden set matrix and then exchanging the 1's for 0's and the 0's for 1's.

Constructing the basis matrices for a k out of k visual threshold scheme is simply a matter of creating two matrices in which each column is a permutation of some number of shares into the total number of shares. For the basis matrix used in encrypting white pixels, only even numbers are permuted into the total number of shares, up to the total number of shares. For the basis matrix used in encrypting black pixels, only odd numbers are used in the permutation.

Finally, the basis matrices for the t out of k visual threshold scheme are created. To do this, we initialize the basis matrices so that all the elements are 0 and then go through each row of each of the basis matrices (their size will be the same as the two k out of k basis matrices). We then look at the corresponding row of the cumulative array matrix, going through each column of that row and checking to see if it holds a 1. If it does, then we logically OR the row of the k out of k basis matrix corresponding to the column we are looking at with the same row of the t out of k basis matrix as the row of the basis matrix we are considering. After each row of the basis matrices is generated in this fashion, they are done and can be used to encrypt the image.

Using these matrices is simply a matter of taking a random permutation of the columns of the matrices for each pixel and encoding the pixel appropriately. For example, in the simple two out of two example used previously, the final basis matrices would look like:

	Black Pixels		White Pixels	
Share #1	0	1	1	0
Share #2	1	0	1	0

If, while going through the image, the program finds a black pixel, it will randomly select one of the columns from the first matrix and then write a black (1) or white (0) pixel to the appropriate share. After this, the same thing is done with the remaining column. If instead, the program finds a white pixel, then the columns simply come from the second matrix instead.

In addition to this standard scheme, there is also an approach used for certain user defined access structures which the standard scheme cannot handle on its own. The scheme essentially splits the access structure into multiple smaller structures, generating basis matrices the same way the normal scheme does, and then combining each of the basis matrices to produce a single pair of basis matrices for the original access structure.

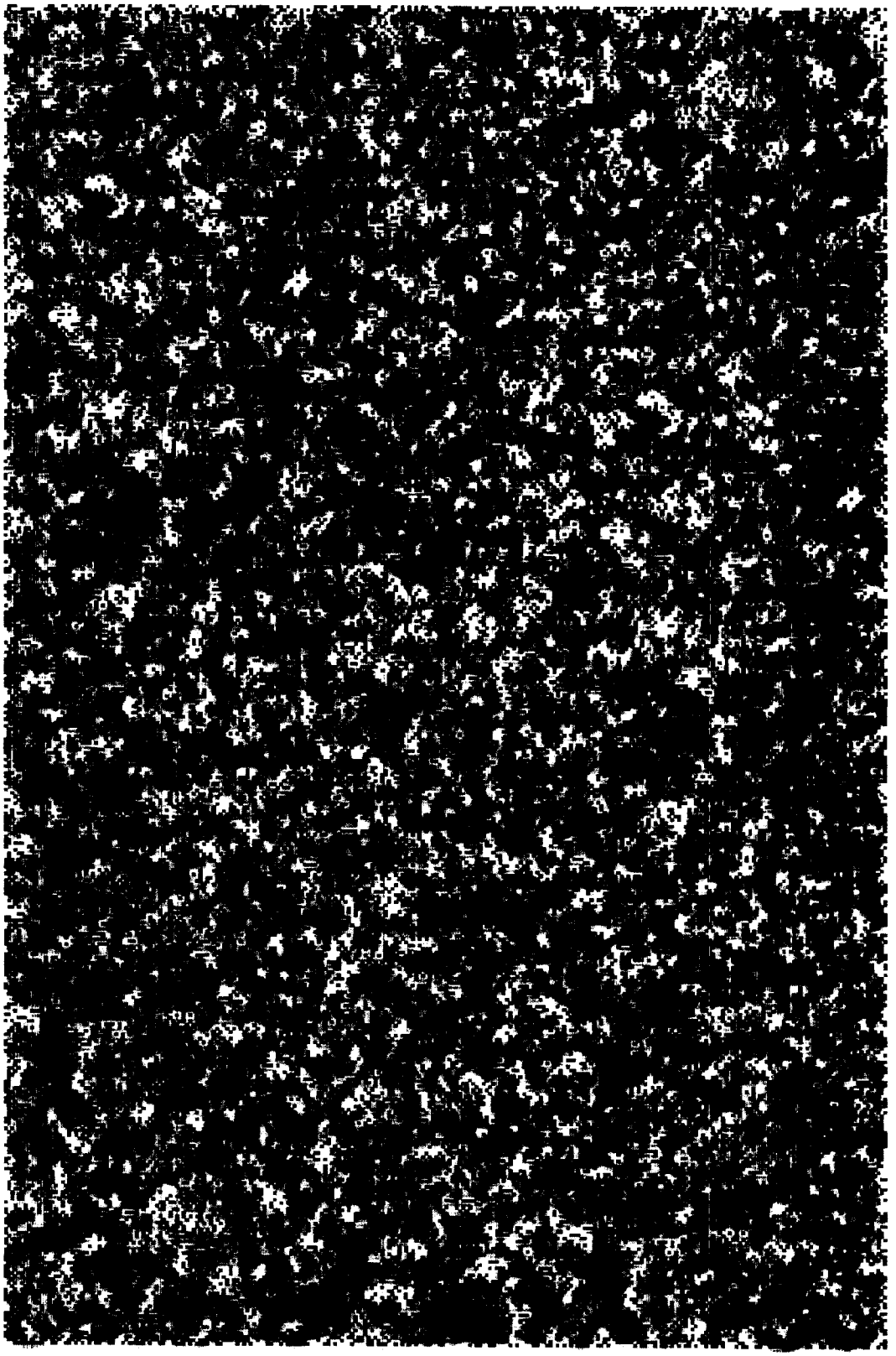
Unfortunately, due to time constraints, certain optimizations to the scheme were not possible. These optimizations include possible methods for increasing the speed of the encryption process as well as adding specific schemes to enhance the contrast in the encrypted images for easier viewing. Also, there are certain encryption configurations which EncryptThis cannot handle due to the lack of certain algorithms. Since these optimizations are not implemented in the program, they are not discussed here.

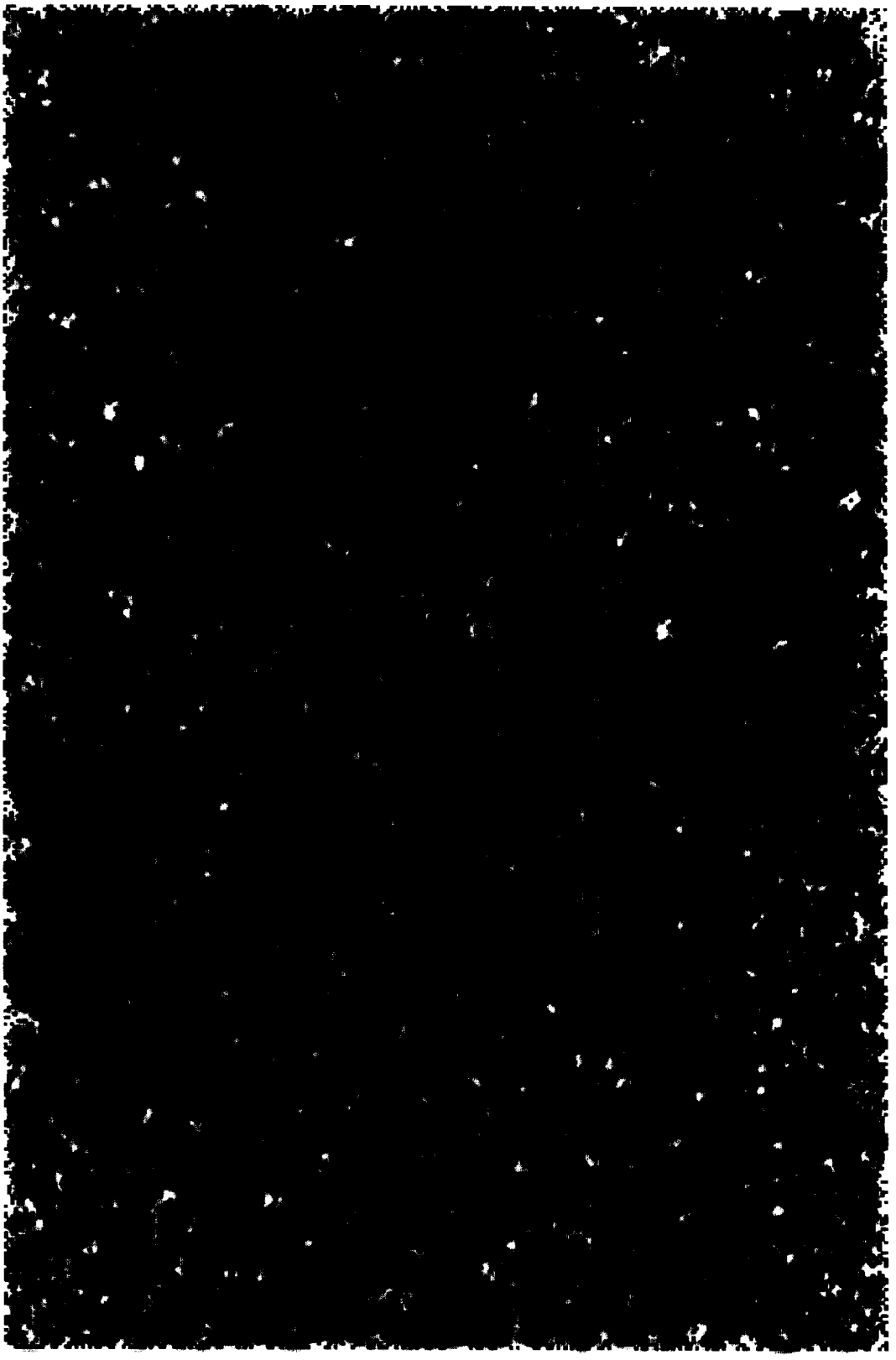
Appendices

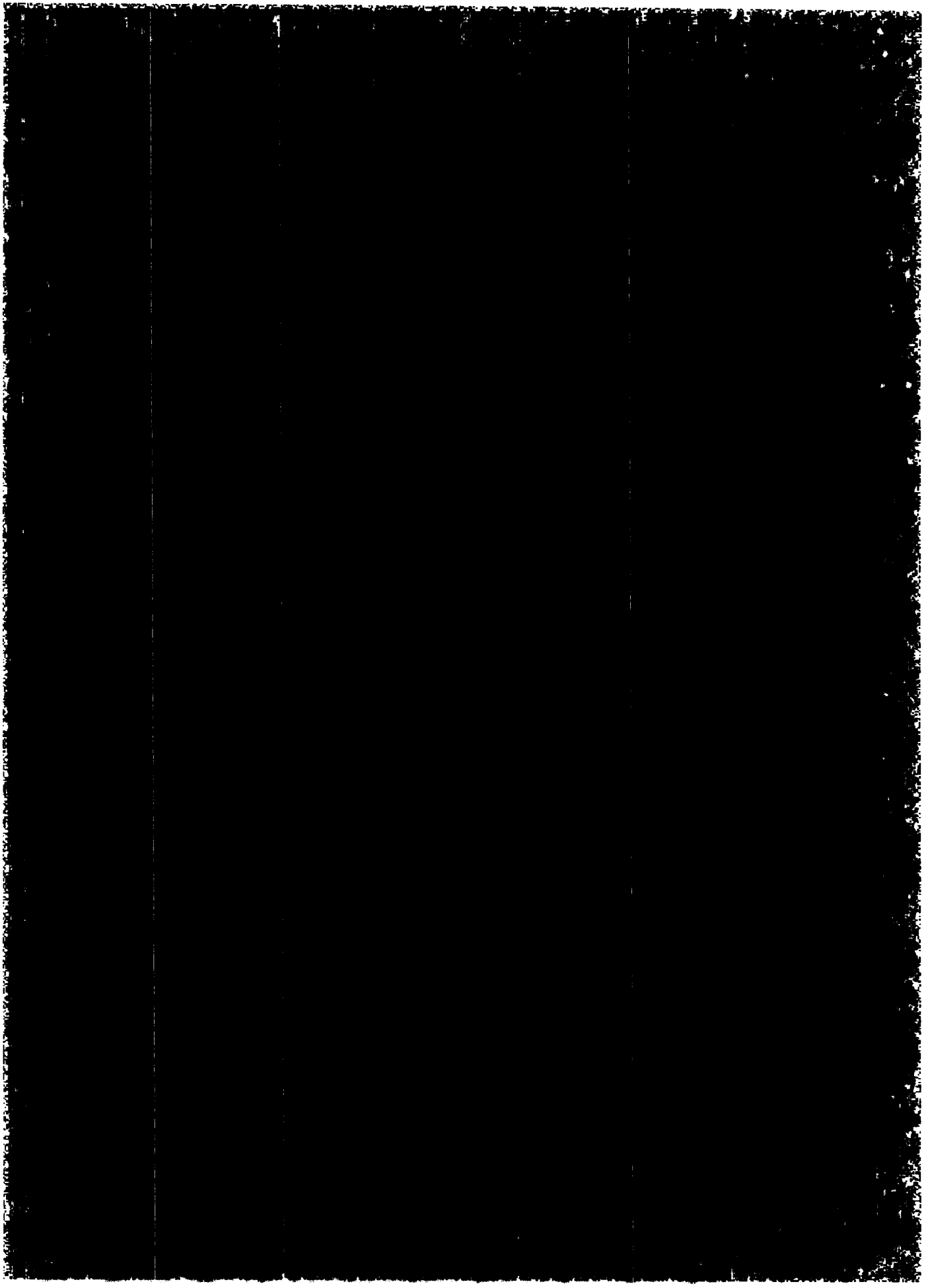
Appendix A – EncryptThis Examples

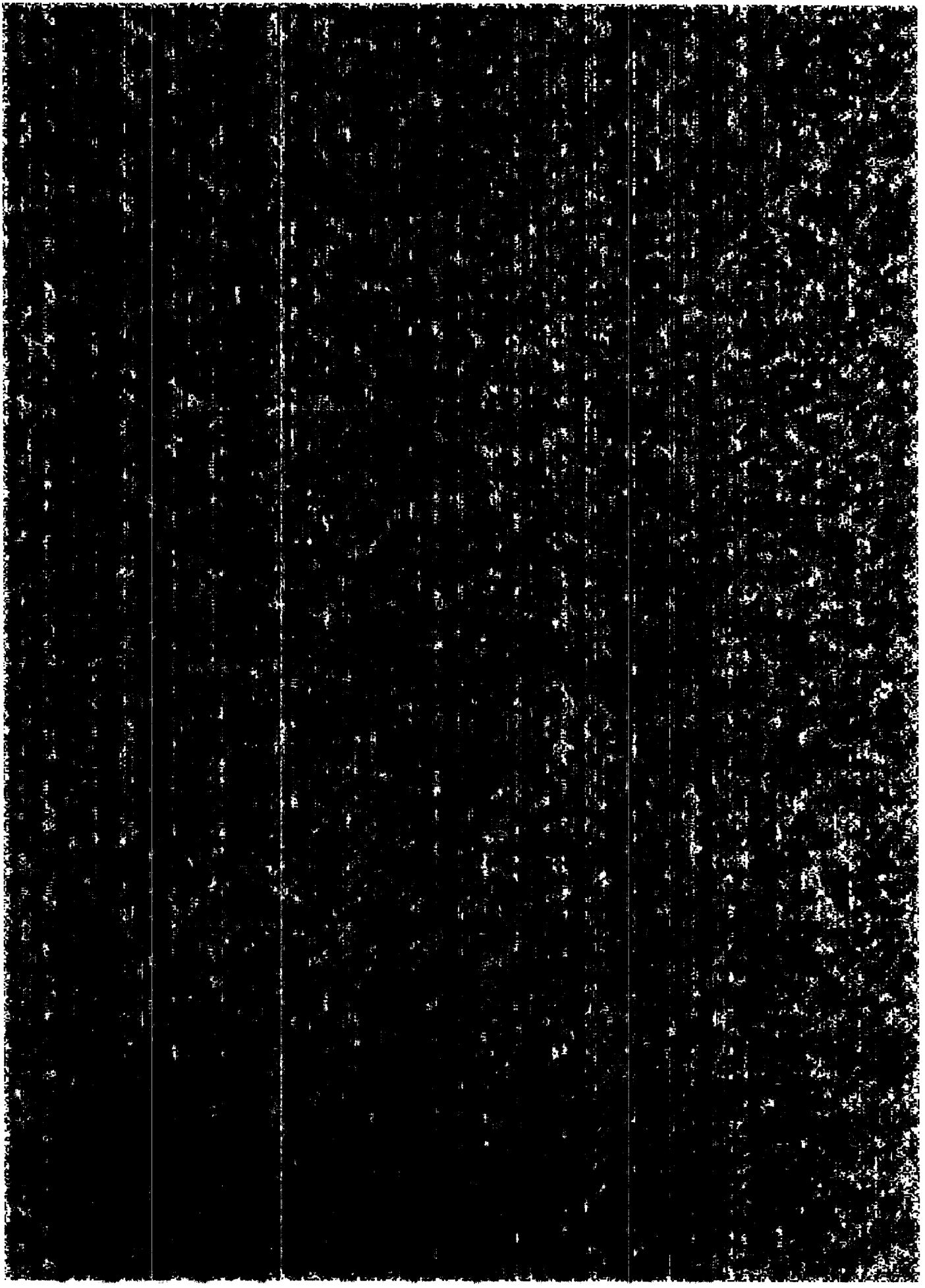
The following pages are two examples of images which have been encrypted into individual shares by EncryptThis. Each black and white image was encrypted using the 2 out of 2 scheme. Each example includes the original image and the shares printed on transparencies which you are free to remove and test.

The first image is a good example of what EncryptThis will encrypt with the best quality. The second image is a more detailed image which does not encrypt well using this encryption scheme.









Appendix B – EncryptThis Source Code and Software

The following pages are the Visual C++ source code used to create the EncryptThis application. After the source code pages, you will find a page holding the EncryptThis software disk which includes the EncryptThis program as well as a few sample images that can be easily encrypted. On the disk you will also find a text file(EncryptThis Readme.txt) which should be viewed for notes on running EncryptThis.

```
#ifndef __BITIMAGE_H
#define __BITIMAGE_H

//
// Copyright (c) 1997,1998 Colosseum Builders, Inc.
// All rights reserved.
//
// Colosseum Builders, Inc. makes no warranty, expressed or implied
// with regards to this software. It is provided as is.
//
// See the README.TXT file that came with this software for information
// on redistribution or send E-mail to info@colosseumbuilders.com
//
// o The user assumes all risk for using this software. The authors of this
//   software shall be liable for no damages of any kind.
//
// o If the source code is distributed then this copyright notice must
//   remain unaltered and any modification must be noted.
//
// o If this code is shipped in binary format the accompanying documentation
//   should state that "this software is based, in part, on the work of
//   Colosseum Builders, Inc."
//

//
//  Title: BitmapImage Class Definitions
//
//  Author:  John M. Miano  miano@colosseumbuilders.com
//
//    The BitmapImage class is intended to be a neutral intermediate format
//    for storing decompressed images. This class can manage 1, 2, 4, 8 or
//    24-bit images. For 24-bit images the data is stored as RGB triples
//    within the main data buffer. For all other types a color map is used
//    and the image data contains indices into the color map. Sample values are
//    assumed to be in the range 0..255.
//
//  Windows Notes:
//
//    For the sake of "efficiency" this class has been optimized for use on
//    the "Windows" family. The folling oddities are a result of
//    "windowsisms":
//
//    o The data for 24-bitmaps is stored in BGR order rather than RGB.
//      To change this for your system redefine "RedOffset", "GreenOffset",
//      and "BlueOffset".
//
//    o For whatever reason, Windows expects bitmaps to be stored bottom
//      up rather than top down. The first row in the bitmap data is the
//      bottom row in the image. To change behavoir this for your system
//      redefine the implementation of the [] operator.
//
//    o Windows expects the length of all image rows to be rounded up to the
//      nearest four bytes. To change this behavior redefine the value for
//     "RowRounding".
//
//  Debugging Notes:
//
//    Two methods for accessing pixel data within the image are implemented
//    by default range checking is only performed on rows. If the
//    preprocessor symbol CHECK_RANGE is defined then range check is
//    performed on columns as well.
//
//    While the abandonment of range checking here is contrary to the
//    principles followed elsewhere, this is a place where the
//    performance benefit is worth the lack of safety.
//

#include <iostream>
#include "datatype.h"

class BitmapImage ;
class BitmapImageCoder ;


typedef void (*PROGRESSFUNCTION)(BitmapImageCoder &coder,
                                 void *data,
```

```
                                    unsigned int currentpass,
                                    unsigned int passcount,
                                    unsigned int progress,
                                    bool &cancel) ;

typedef void (*IMAGEPROGRESSFUNCTION)(BitmapImage &image,
                                      void *data,
                                      unsigned int currentpass,
                                      unsigned int passcount,
                                      unsigned int progress,
                                      bool &cancel) ;

class BitmapImageCoder
{
public:
  BitmapImageCoder () : progress_function (NULL), progress_data (NULL) {}
  BitmapImageCoder (const BitmapImageCoder &source) ;
  virtual ~BitmapImageCoder () {}
  BitmapImageCoder &operator=(const BitmapImageCoder &source) ;

  void SetProgressFunction (PROGRESSFUNCTION, void *) ;
  // Function to force an update of image data.
  virtual void UpdateImage () {}
protected:
  void Initialize () ;
  void DoCopy (const BitmapImageCoder &source) ;
  PROGRESSFUNCTION progress_function ;
  void *progress_data ;
} ;

inline BitmapImageCoder::BitmapImageCoder (const BitmapImageCoder &source)
{
  DoCopy (source) ;
  return ;
}

inline BitmapImageCoder &BitmapImageCoder::operator=(
                                const BitmapImageCoder &source)
{
  DoCopy (source) ;
  return *this ;
}

inline void BitmapImageCoder::DoCopy (const BitmapImageCoder &source)
{
  progress_function = source.progress_function ;
  progress_data = source.progress_data ;
  return ;
}

inline void BitmapImageCoder::SetProgressFunction (PROGRESSFUNCTION func,
                                                   void *data)
{
  progress_function = func ;
  progress_data = data ;
  return ;
}

class BitmapImageDecoder : public BitmapImageCoder
{
public:
  virtual ~BitmapImageDecoder () {}
  virtual void ReadImage (std::istream &, BitmapImage &) = 0 ;
} ;

class BitmapImageEncoder : public BitmapImageCoder
{
public:
  virtual ~BitmapImageEncoder () {}
  virtual void WriteImage (std::ostream &, BitmapImage &) = 0 ;
} ;

class BitmapImage
{
public:
#if defined (CHECK_RANGE)
```

```
    // The Row class is used to implement range checking on columns. A Row
    // object represents a row if image data.
    class Row
    {
    public:
      UBYTE1 &operator[](unsigned int) ;
    private:
      Row (UBYTE1 *data, unsigned int length) ;
      UBYTE1 *row_data ;
      unsigned int row_length ;
      friend class BitmapImage ;
    } ;
  #endif

    // Definition of the color map used by bitmaps of other than 24-bits.
    struct ColorMapEntry
    {
      UBYTE1 blue ;
      UBYTE1 green ;
      UBYTE1 red ;
    } ;

    enum { RedOffset=2, GreenOffset=1, BlueOffset=0 } ;

    // Required Member Functions
    BitmapImage () ;
    BitmapImage (const BitmapImage &) ;
    virtual ~BitmapImage () ;
    BitmapImage &operator=(const BitmapImage &) ;

    // Function to allocate space for to store an image.
    void SetSize (unsigned int cc,    // Color Count
                  unsigned int bits,  // Bit Count
                  unsigned int ww,    // Width
                  unsigned int hh) ;  // Height

    // Function to retrieve entries in the color map.
    ColorMapEntry &ColorMap (unsigned int index) ;
    ColorMapEntry ColorMap (unsigned int index) const ;

    // [] returns image data bytes.
  #if defined (CHECK_RANGE)
    Row operator[](unsigned int) const ;
  #else
    UBYTE1 *operator[](unsigned int) const ;
  #endif
    // Function to reset the image to empty
    void Clear () ;

    // Function to return information about the image.
    UBYTE1 *ImageData() ;
    unsigned int Width () const ;
    unsigned int Height () const ;
    unsigned int BitCount () const ;
    unsigned int ColorCount () const ;
    void GetRGB (unsigned int row, unsigned int col,
                 UBYTE1 &red, UBYTE1 &green, UBYTE1 &blue) const ;

    void EightBitQuantization (const BitmapImage &) ;

    void SetProgressFunction (IMAGEPROGRESSFUNCTION, void *) ;

    // Number of bytes to round each row to. On Windows this should be 4.
    enum { RowRounding = 4 } ;

    unsigned int BytesPerRow () const ;
  protected:
    void Initialize () ;
    void DoCopy (const BitmapImage &) ;

  private:
    // Width in bytes of each row. For 24-bit images this value will
    // always be larger than the width of the image. For 8-bit images
    // it may be larger if the row width is rounded up.
    unsigned int row_width ;
    unsigned int bit_count ;      // Number of bits (1, 2, 4, 8, or 24)
```

```
    unsigned int image_width ;
    unsigned int image_height ;
    // Image data arranged left to right, top to bottom. For 24-bit images
    // there are 3 bytes per pixel representing the color to display in RGB order.
    // For all others the data is in index into the color map. 1, 2, and 4-bit
    // image row widths are rounded up to the nearest byte.
    unsigned char *image_data ;

    unsigned int color_count ;    // Number of entries in the color map.
    ColorMapEntry *color_map ;    // Color map for 8-bit image

    // Interal function to state class variables to a known state.
    void ClearData () ;

    struct ColorUsage
    {
      UBYTE1 colors [3] ;
      UBYTE4 usage ;
      ColorUsage *next [3] ;
    } ;

    struct ColorUsageTable
    {
      ColorUsage *lists [256][3] ;
      unsigned int color_count ;
    } ;

    struct ColorArea
    {
      struct
      {
        UBYTE1 low ;
        UBYTE1 high ;
      } color_values [3] ;
      unsigned int color_count ;
      unsigned int pixel_count ;
    }  ;

    int LargestColorRange (ColorArea &area) ;
    void AddColor (UBYTE1 red, UBYTE1 green, UBYTE1 blue) ;
    void SplitAreaInHalf (unsigned int depth, unsigned int retrydepth, unsigned int area, unsigned i
  nt splitcolor) ;
    void CreateColor (unsigned int color) ;
    ColorUsage *FindColor (UBYTE1 red, UBYTE1 green, UBYTE1 blue) ;
    void FindColorUsage (const BitmapImage &image) ;
    void FreeColorQuantizationData () ;
    unsigned int QuantizedColor (UBYTE1 red, UBYTE1 green, UBYTE1 blue) ;
    void QuantizeSourceImage (const BitmapImage &image) ;

    ColorUsageTable *color_usage ;
    ColorArea *color_areas ;
    unsigned int color_area_count ;

    void CallProgressFunction (unsigned int, unsigned int, unsigned int) ;
    IMAGEPROGRESSFUNCTION progress_function ;
    void *progress_data ;
  } ;

  inline UBYTE1 *BitmapImage::ImageData()
  {
    return image_data ;
  }

  inline unsigned int BitmapImage::Width () const
  {
    return image_width ;
  }

  inline unsigned int BitmapImage::Height () const
  {
   return image_height ;
  }

  inline unsigned int BitmapImage::BitCount () const
  {
    return bit_count ;
```

```
}

inline unsigned int BitmapImage::ColorCount () const
{
  return color_count ;

#if ! defined (CHECK_RANGE)
inline UBYTE1 *BitmapImage::operator[](unsigned int xx)const
{
  return &image_data [(image_height - xx - 1) * row_width] ;
}
#endif

inline unsigned int BitmapImage::BytesPerRow () const
{
  return row_width ;
}


#endif
```

```
//
// Copyright (c) 1997 Colosseum Builders, Inc.
// All rights reserved.
//
// Colosseum Builders, Inc. makes no warranty, expressed or implied
// with regards to this software. It is provided as is.
//
// Permission to use, redistribute, and copy this file is granted
// without a fee so long as as the following conditions are adhered to:
//
// o The user assumes all risk for using this software. The authors of this
//   software shall be liable for no damages of any kind.
//
// o If the source code is distributed then this copyright notice must
//   remain unaltered and any modification must be noted.
//
// o If this code is shipped in binary format the accompanying documentation
//   should state that "this software is based, in part, on the work of
//   Colosseum Builders, Inc."
//

//
//  Title:  Bitmap Image Class Implementation
//
//  Author:  John M. Miano miano@colosseumbuilders.com
//
#include "bitimage.h"
#include "grexcept.h"
#include "stdafx.h"

// Function to round a row width up to the nearest multiple of
// RowRounding. Windows expects rows to be a length that is a multiple of
// four.

static inline int SQUARE (int xx)
{
  return xx * xx ;
}

static inline unsigned int RoundRow (unsigned int width)
{
  unsigned int result = (width + BitmapImage::RowRounding - 1)
                      & ~(BitmapImage::RowRounding - 1) ;
  return result ;
}

//
//  Description:
//
//    Default Constructor
//
BitmapImage::BitmapImage ()
{
  Initialize () ;
  return ;
}

//
//  Description:
//
//    Copy Constructor
//
BitmapImage::BitmapImage (const BitmapImage &source)
{
  Initialize () ;
  DoCopy (source) ;
  return ;
}

//
//  Description:
//
//    Assignment Operator
//
//  Parameters:
//    source:  The object to copy
```

```
//
BitmapImage &BitmapImage::operator=(const BitmapImage &source)
{
  delete [] color_map ;
  delete [] image_data ;
  DoCopy (source) ;
  return *this ;
}

//
//  Description:
//
//    Common initialization function.
//
void BitmapImage::Initialize ()
{
  ClearData () ;
  progress_function = NULL ;
  progress_data = NULL ;
  return ;
}

//
//  Description:
//
//    Class Destructor
//
BitmapImage::~BitmapImage ()
{
  delete [] color_map ;
  delete [] image_data ;
  return ;
}

//
//  Description:
//
//    Common function for resetting an object to a known state.
//
void BitmapImage::ClearData ()
{
  bit_count = 0 ;
  image_width = 0 ;
  image_height = 0 ;
  color_count = 0 ;
  color_map = NULL ;
  image_data = NULL ;
  color_usage = NULL ;
  color_areas = NULL ;
  color_area_count = 0 ;

  return ;
}

//
//  Description:
//
//    Common copy function for use by the copy constructor and assignment
//    operator.
//
//  Parameters:
//    source:  The object to copy
//
void BitmapImage::DoCopy (const BitmapImage &source)
{
  progress_function = source.progress_function ;
  progress_data = source.progress_function ;

  bit_count = source.bit_count ;
  image_width = source.image_width ;
  image_height = source.image_height ;
  color_count = source.color_count ;
  color_map = NULL ;
  image_data = NULL ;

  color_usage = NULL ;
```

```
    color_areas = NULL ;
    color_area_count = 0 ;

    // Only copy the image data if the size values are valid.
    if (image_width > 0 && image_height > 0 && bit_count > 0
        && (bit_count == 24 || color_count != 0))
    {
      unsigned int bitwidth ;
      unsigned int bytecount ;
      switch (bit_count)
      {
      case 1:
      case 2:
      case 4:
      case 8:
        color_map = new ColorMapEntry [color_count] ;
        memcpy (color_map,
                source.color_map,
                sizeof (ColorMapEntry) * color_count) ;
        bitwidth = bit_count * image_width ;
        row_width = RoundRow ((bitwidth + 7)/8) ;

        bytecount = row_width * image_height ;
        image_data = new UBYTE1 [bytecount] ;
        memcpy (image_data, source.image_data, bytecount) ;
        break ;

      case 24:
        row_width = RoundRow (3 * image_width) ;
        image_data = new UBYTE1 [row_width * image_height] ;
        memcpy (image_data, source.image_data, row_width * image_height) ;
        break ;
      default:
        if (image_width != 0 ||image_height != 0)
          throw EInvalidBitCount () ;
      }
    }
    return ;
}

  #if defined (CHECK_RANGE)
  //
  //  Description:
  //
  //    Row Class Constructor.  The row class represents a single
  //    row of image data.
  //
  //  Parameters:
  //    data: A pointer to the row's data.
  //    length: The row length
  //
  BitmapImage::Row::Row (UBYTE1 *data, unsigned int length)
  {
    row_data = data ;
    row_length = length ;
    return ;
  }

  //
  //  Description:
  //
  //    Row class [] operator. This operator returns the data
  //    value at a given point offset in the row.
  //
  //  Parameters:
  //    index:  The row offset
  //
  //  Return Value:
  //    The data value in the row at the specified offset
  //
  UBYTE1 &BitmapImage::Row::operator[](unsigned int index)
  {
    if (index >= row_length)
      throw ESubscriptOutOfRange () ;

    return row_data [index] ;
```

```
}

//
//  Description:
//
//    This function returns a pointer to the Nth row of the image's data.
//    It is set up to return the rows in reverse order as Windows expects
//    them so that other software does not have to deal with such wierdness.
//
//  Parameters:
//    xx: The row index
//
//  Return Value:
//    A Row object that describes the image row.
//
BitmapImage::Row BitmapImage::operator[](unsigned int xx)const
{
  // In Windows bitmaps are stored bass ackwards.
  if (xx >= image_height)
    throw ESubscriptOutOfRange () ;
  return Row (&image_data [(image_height - xx - 1) * row_width],
              row_width) ;
}
#endif

//
//  Description:
//
//    This function allocates space to hold an image of the specified size.
//    The colormap (if used) and the image data are all set to zeros.
//
//  Parameters:
//    cc: Number of colors. Ignored for 24-bit bitmaps
//    bits: Number of bits per pixel.
//    ww, hh: Bitmap size
//
void BitmapImage::SetSize (unsigned int cc,     // Color Count
                           unsigned int bits,   // Data Size in Bits
                           unsigned int ww,     // Width
                           unsigned int hh)     // Height
{
  // Get rid of any existing image.
  delete [] color_map ;
  delete [] image_data ;
  ClearData () ;

  switch (bits)
  {
  case 1:
  case 2:
  case 4:
  case 8:
    {
      bit_count = bits ;
      color_count = cc ;
      image_width = ww ;
      image_height = hh ;

      color_map = new ColorMapEntry [color_count] ;
      memset (color_map, 0, sizeof (ColorMapEntry) * color_count) ;
      unsigned int bitsize = bit_count * image_width ;
      row_width = RoundRow ((bitsize + 7)/8) ;
      unsigned int bytecount = row_width * image_height ;
      image_data = new UBYTE1 [bytecount] ;
      memset (image_data, 0, bytecount) ;
    }
    break ;

  case 24:
    {
      bit_count = bits ;
      color_count = cc ;
      image_width = ww ;
      image_height = hh ;
      row_width = RoundRow (3 * image_width) ;
      image_data = new UBYTE1 [row_width * image_height] ;
```

```
      memset (image_data, 0, row_width * image_height) ;
    }
    break ;
  default:
    throw EInvalidBitCount () ;
  }
  return ;
}

//
//  Description:
//
//    This function returns a reference to the Nth colormap entry.
//
//  Parameters:
//    index:  The index of the color map entry 0..ColorCount () -1.
//
//  Return Value:
//    The color map entry.
//
BitmapImage::ColorMapEntry &BitmapImage::ColorMap (unsigned int index)
{
  if (index >= color_count)
    throw ESubscriptOutOfRange () ;

  return color_map [index] ;
}

//
//  Description:
//
//    This function returns a reference to the Nth colormap entry.
//
//    This is a const version of the previous function.
//
//  Parameters:
//    index:  The index of the color map entry 0..ColorCount () -1.
//
//  Return Value:
//    The color map entry.
//
BitmapImage::ColorMapEntry BitmapImage::ColorMap (unsigned int index) const
{
  if (index >= color_count)
    throw ESubscriptOutOfRange () ;

  return color_map [index] ;
}

//
//  Description:
//
//    This function clears out the image.
//
void BitmapImage::Clear ()
{
  delete [] color_map ;
  delete [] image_data ;
  ClearData () ;
  return ;
}

//
//  Description:
//
//   This function returns the RGB values for a pixel in the bitmap at the
//   point [row,col] where row=[0..height-1] and col=[0..width-1].
//
//   This function allows a caller to get the RGB values for 1, 2, 4, 6,
//   and 24-bit bitmaps using the same method.
//
//  Parameters:
//    row, col: The position in the image to return data from
//    red, green, blue:  The color value at the specified position
//
void BitmapImage::GetRGB (unsigned int row, unsigned int col,
```

```
                                UBYTE1 &red, UBYTE1 &green, UBYTE1 &blue) const
{
  if (row >= image_height && col >= image_width)
    throw ESubscriptOutOfRange () ;

  switch (bit_count)
  {
    unsigned int index ;
    unsigned int offset ;
  case 1:
    offset = col / 8  ;
    index = (((*this)[row][offset] >> (7 - (col % 8))) & 0x1) ;
    red = color_map [index].red ;
    green = color_map [index].green ;
    blue = color_map [index].blue ;
    break ;

  case 2:
    offset = col / 4 ;
    index = (((*this)[row][offset] >> (2 * (3 - (col % 4)))) & 0x3) ;
    red = color_map [index].red ;
    green = color_map [index].green ;
    blue = color_map [index].blue ;
    break ;

  case 4:
    offset = col / 2 ;
    if (col % 2 == 0)
      index = ((*this)[row][offset] & 0xF0) >> 4 ;
    else
      index = ((*this)[row][offset] & 0x0F) ;

    red = color_map [index].red ;
    green = color_map [index].green ;
    blue = color_map [index].blue ;
    break ;

  case 8:
    red = color_map [(*this)[row][col]].red ;
    green = color_map [(*this)[row][col]].green ;
    blue = color_map [(*this)[row][col]].blue ;
    break ;

  case 24:
    red = (*this)[row][3 * col + RedOffset] ;
    green = (*this)[row][3 * col + GreenOffset] ;
    blue = (*this)[row][3 * col + BlueOffset] ;
    break ;
  default:
    throw EInvalidBitCount () ;
  }
  return ;
}

//
// Description:
//    This sets the progress function for the image.
//
// Parameters:
//    function:   The progress function
//    data:       The call progress data
//
void BitmapImage::SetProgressFunction (
                                IMAGEPROGRESSFUNCTION function,
                                void *data)
{
  progress_function = function ;
  progress_data = data ;
  return ;
}

//
// Description:
//    This function calls the progression function.
//
// Parameters:
```

```
//    percent:    % complete
//    pass:       Current Pass
//    passcount:  Number of passes
//
void BitmapImage::CallProgressFunction (unsigned int percent,
                                        unsigned int pass,
                                        unsigned int passcount)
{
  if (progress_function == NULL)
    return ;
  if (percent > 100)
    percent = 100 ;

  bool cancel = false ;
  progress_function (*this, progress_data, pass, passcount, percent, cancel) ;
  if (cancel)
    throw EGraphicsAbort () ;
  return ;
}

//
// Color Quantization Routines
//
//  Since I have received many requests for color quantization I have
//  whipped this up. I have to admit to knowing nothing about color
//  quantization (I have always had systems that did not need it). The
//  only techniques for quantization that I had been familiar with are
//  scaling color values and having a fixed color space. However, I thought
//  1-pass methods such as those would be too cude for someone of my
//  programming skill.
//
//  What I have tried instead is to use two passes. What we do is create
//  a 3-dimensional hash table of color values. The we keep dividing the
//  colorspace in half until we have set of color ranges.
//
//  Hey, it's probably not all that efficient but it's the best I could come
//  up with in an evening.
//
//

//
//  Description:
//    This function looks up the color entry in the color hash table
//    for a specified color value.
//
//  Parameters:
//    red, green, blue:  The color value to search for.
//
BitmapImage::ColorUsage *BitmapImage::FindColor (UBYTE1 red,
                                                 UBYTE1 green,
                                                 UBYTE1 blue)
{
  if (color_usage->lists [red][RedOffset] == NULL
      || color_usage->lists [green][GreenOffset] == NULL
      || color_usage->lists [blue][BlueOffset] == NULL)
  {
    return NULL ;
  }

  for (ColorUsage *entry = color_usage->lists [red][RedOffset] ;
       entry != NULL ;
       entry = entry->next [RedOffset])
  {
    if (entry->colors [BlueOffset] == blue
        && entry->colors [GreenOffset] == green)
    {
      return entry ;
    }
  }
  return NULL ;
}

//
// Description:
//  This function adds a new color value to the color hash table.
//
```

```
//  Parameters:
//    red, green, blue:  The color value to search for.
//

void BitmapImage::AddColor (UBYTE1 red, UBYTE1 green, UBYTE1 blue)
{
  // Create the new color entry.
  ColorUsage *entry = new ColorUsage ;
  memset (entry, 0, sizeof (*entry)) ;
  entry->usage = 1 ;
  entry->colors [RedOffset] = red ;
  entry->colors [GreenOffset] = green ;
  entry->colors [BlueOffset] = blue ;

  // Add the new entry to each hash chain.
  if (color_usage->lists [red][RedOffset] == NULL)
  {
    color_usage->lists [red][RedOffset] = entry ;
  }
  else
  {
    entry->next [RedOffset] = color_usage->lists [red][RedOffset] ;
    color_usage->lists [red][RedOffset] = entry ;
  }
  if (color_usage->lists[green][GreenOffset] == NULL)
  {
    color_usage->lists [green][GreenOffset] = entry ;
  }
  else
  {
    entry->next [GreenOffset] = color_usage->lists [green][GreenOffset] ;
    color_usage->lists [green][GreenOffset] = entry ;
  }
  if (color_usage->lists [blue][BlueOffset] == NULL)
  {
    color_usage->lists [blue][BlueOffset] = entry ;
  }
  else
  {
    entry->next [BlueOffset] = color_usage->lists [blue][BlueOffset] ;
    color_usage->lists [blue][BlueOffset] = entry ;
  }

  ++ color_usage->color_count ;
  return ;
}

//
// Description:
//  This function converts a 24-bit image to 8-bit.
//
// Parameters:
//  image: The image to convert.
//
void BitmapImage::EightBitQuantization (const BitmapImage &image)
{
  // If this is not a 24-bit image then there is no need to quantize.
  // Instead, we make this a copy operation.
  if (image.bit_count != 24)
  {
    *this = image ;
    return ;
  }

  progress_function = image.progress_function ;
  progress_data = image.progress_function ;

  // Allocate space for the image.
  SetSize (256, 8, image.image_width, image.image_height) ;

  // Allocate temporary structures used for color quantization.
  color_usage = new ColorUsageTable ;
  memset (color_usage, 0, sizeof (*color_usage)) ;
  color_areas = new ColorArea [256] ;

  try
```

```
{
  FindColorUsage (image) ;
}
catch (EGraphicsAbort &)
{
  FreeColorQuantizationData () ;
  return ;
}
catch (...)
{
  FreeColorQuantizationData () ;
  throw ;
}

// Set the first (zero'th) area to the entire color space.
color_areas [0].color_values [RedOffset].low = 0 ;
color_areas [0].color_values [RedOffset].high = 255 ;
color_areas [0].color_values [GreenOffset].low = 0 ;
color_areas [0].color_values [GreenOffset].high = 255 ;
color_areas [0].color_values [BlueOffset].low = 0 ;
color_areas [0].color_values [BlueOffset].high = 255 ;
color_areas [0].pixel_count = image_height * image_width ;
color_areas [0].color_count = color_usage->color_count ;
color_area_count = 1 ;

// Divide the color area in half.
SplitAreaInHalf (7, // Depth
                 0, // Retry Count
                 0, // Area Number
                 RedOffset) ; // Split Color

// It is possible that some of the areas could not be divided using the
// previous process. If we have some remaining colors we try to assign them
// to the blocks with the largest size in the colorspace.
while (color_area_count < 256)
{
  int cb = 0 ;
  // Search for the largest colorspace area.
  unsigned int value
                 = SQUARE (color_areas [cb].color_values [RedOffset].high -
                           color_areas [cb].color_values [RedOffset].low)
                 + SQUARE (color_areas [cb].color_values [GreenOffset].high -
                           color_areas [cb].color_values [GreenOffset].low)
                 + SQUARE (color_areas [cb].color_values [BlueOffset].high -
                           color_areas [cb].color_values [BlueOffset].low)
                 * color_areas [cb].color_count ;
  for (unsigned int ii = 1 ; ii < color_area_count ; ++ ii)
  {
    if (color_areas [ii].color_count > 1)
    {
      unsigned int newvalue
                 = SQUARE (color_areas [ii].color_values [RedOffset].high -
                           color_areas [ii].color_values [RedOffset].low)
                 + SQUARE (color_areas [ii].color_values [GreenOffset].high -
                           color_areas [ii].color_values [GreenOffset].low)
                 + SQUARE (color_areas [ii].color_values [BlueOffset].high -
                           color_areas [ii].color_values [BlueOffset].low)
                 * color_areas [ii].color_count ;
      if (newvalue > value)
      {
        value = newvalue ;
        cb = ii ;
      }
    }
  }
  // If we have not colors to divide then stop.
  if (color_areas [cb].color_count == 1)
    break ;

  // Split this color block in half.
  SplitAreaInHalf (0, // Depth
                   0, // Retry Count
                   cb, // Area Number
                   LargestColorRange (color_areas [cb])) ; // Split Color
}
```

```
    for (unsigned int ii = 0 ; ii < color_area_count ; ++ ii)
    {
      CreateColor (ii) ;
    }
    // Assign colors to the image.
    try
    {
      QuantizeSourceImage (image) ;
    }
    catch (EGraphicsAbort &)
    {
      FreeColorQuantizationData () ;
      return ;
    }
    catch (...)
    {
      FreeColorQuantizationData () ;
      throw ;
    }

    FreeColorQuantizationData () ;
    return ;
  }

  //
  // Description:
  //    This function finds the colors that are used and their frequency
  //    within a source image.
  //
  // Parameters:
  //    image:   The source image.
  //
  void BitmapImage::FindColorUsage (const BitmapImage &image)
  {
    // Create a color entry for each distinct color.
    const unsigned int climit = image_width * 3 ;
    for (unsigned int rr = 0 ; rr < image_height ; ++ rr)
    {
      UBYTE1 *rowdata = &image.image_data [rr * image.row_width] ;
      for (unsigned int cc = 0 ; cc < climit ; cc += 3 )
      {
        UBYTE1 red, green, blue ;
        red = rowdata [cc + RedOffset] ;
        green = rowdata [cc + GreenOffset]  ;
        blue = rowdata [cc + BlueOffset] ;

        // If the color already exists in the table just increment
        // its usage count. Otherwise add a new color entry for it.
        ColorUsage *entry = FindColor (red, green, blue) ;
        if (entry == NULL)
        {
          AddColor (red, green, blue) ;
        }
        else
        {
          ++ entry->usage ;
        }
      }
      CallProgressFunction (100 * rr/image_height, 1, 2) ;
    }
    CallProgressFunction (100, 1, 2) ;

    return ;
  }

  //
  //  Description:
  //
  //    This function frees all the dynamic data allocated during
  //    color quantization.
  //
  void BitmapImage::FreeColorQuantizationData ()

    // Get rid of al the temporary storage.
    for (unsigned int ii = 0 ; ii < 256 ; ++ ii)
    {
```

```
      ColorUsage *next ;
      for (ColorUsage *entry = color_usage->lists [ii][RedOffset] ;
           entry != NULL ;
           entry = next)
      {
         next = entry->next [RedOffset] ;
         delete entry ;
      }
    }

    delete color_usage ; color_usage = NULL ;
    delete [] color_areas ;
    return ;
  }

  //
  //  Description:
  //
  //    This function divides an area of the colorspace in half.
  //
  //  Parameters:
  //    depth:        The search depth
  //    retrydepth:   The number of retries
  //    areaid:       The area to split
  //    splitcolor:   The color to split on.
  //
  void BitmapImage::SplitAreaInHalf (unsigned int depth,
                                     unsigned int retrydepth,
                                     unsigned int areaid,
                                     unsigned int splitcolor)
  {

    if (color_areas [areaid].color_count == 1)
    {
      return ;
    }
    else if (color_areas [areaid].color_values [splitcolor].high
        == color_areas [areaid].color_values [splitcolor].low)
    {
      if (retrydepth < 2)
      {
        SplitAreaInHalf (depth, retrydepth + 1, areaid, (splitcolor + 1) % 3) ;
      }
      return ;
    }

    unsigned int c1 = (splitcolor + 1) % 3 ;
    unsigned int c2 = (splitcolor + 2) % 3 ;

    unsigned int splitsize = color_areas [areaid].pixel_count / 2 ;
    unsigned int splitpixelcount = 0 ;
    unsigned int splitcolorcount = 0 ;
    unsigned int newlimit ;
    unsigned int newpixelcount ;
    unsigned int newcolorcount ;

    for (newlimit = color_areas [areaid].color_values [splitcolor].low ;
         newlimit <= color_areas [areaid].color_values [splitcolor].high ;
         ++ newlimit)
    {
      newpixelcount = 0 ;
      newcolorcount = 0 ;
      for (ColorUsage *entry = color_usage->lists [newlimit][splitcolor] ;
           entry != NULL ;
           entry = entry->next [splitcolor])
      {
        if (entry->colors [c1] >= color_areas [areaid].color_values [c1].low
            && entry->colors [c1] <= color_areas [areaid].color_values [c1].high
            && entry->colors [c2] >= color_areas [areaid].color_values [c2].low
            && entry->colors [c2] <= color_areas [areaid].color_values [c2].high)
        {
          newpixelcount += entry->usage ;
          ++ newcolorcount ;
        }
      }
```

```
    if (newcolorcount == color_areas [areaid].color_count)
    {
      // There is no way to split using this color.
      if (retrydepth < 2)
      {
        SplitAreaInHalf (depth, retrydepth + 1, areaid, (splitcolor + 1) % 3) ;
      }
      return ;
    }
    else if (newcolorcount > color_areas [areaid].color_count)
    {
      throw EGraphicsException ("INTERNAL ERROR - Quantization area color count invalid") ;
    }

    if (splitpixelcount + newpixelcount >= splitsize)
    {
      if (splitpixelcount + newpixelcount != color_areas [areaid].pixel_count)
      {
        splitpixelcount += newpixelcount ;
        splitcolorcount += newcolorcount ;
      }
      else
      {
        -- newlimit ;
      }
      color_areas [color_area_count] = color_areas [areaid] ;
      color_areas [color_area_count].pixel_count = color_areas [areaid].pixel_count - splitpixelco
unt ;
      color_areas [color_area_count].color_count = color_areas [areaid].color_count - splitcolorco
unt ;
      color_areas [color_area_count].color_values [splitcolor].low = newlimit + 1 ;
      ++ color_area_count ;

      color_areas [areaid].color_values [splitcolor].high = newlimit ;
      color_areas [areaid].pixel_count = splitpixelcount ;
      color_areas [areaid].color_count = splitcolorcount ;

      if (depth > 0)
      {
        SplitAreaInHalf (depth - 1, 0, color_area_count - 1, LargestColorRange (color_areas [color
_area_count-1])) ;
        SplitAreaInHalf (depth - 1, 0, areaid, LargestColorRange (color_areas [areaid])) ;
      }
      return ;
    }
    else
    {
      splitpixelcount += newpixelcount ;
      splitcolorcount += newcolorcount ;
    }
  }
  throw EGraphicsException ("INTERNAL ERROR - Quantization area pixel count invalid") ;
}

//
// Description:
//    This function creates a color from a color area then maps the colors
//    in the source image to the new color map.
//
// Parameters:
//    color:   The new color index value
//
void BitmapImage::CreateColor (unsigned int color)
{
  unsigned int red = 0 ;
  unsigned int green = 0 ;
  unsigned int blue = 0 ;

  const int c0 = RedOffset ;
  const int c1 = GreenOffset ;
  const int c2 = BlueOffset ;

  unsigned int itemcount = 0 ;
  for (unsigned int cc = color_areas [color].color_values [c0].low ;
       cc <= color_areas [color].color_values [c0].high ;
       ++ cc)
```

```
    {
      for (ColorUsage *entry = color_usage->lists [cc][c0] ;
           entry != NULL ;
           entry = entry->next [c0])
      {
        if (entry->colors [c1] >= color_areas [color].color_values [c1].low
            && entry->colors [c1] <= color_areas [color].color_values [c1].high
            && entry->colors [c2] >= color_areas [color].color_values [c2].low
            && entry->colors [c2] <= color_areas [color].color_values [c2].high)
        {
          red += entry->colors [RedOffset] * entry->usage ;
          green += entry->colors [GreenOffset] * entry->usage ;
          blue += entry->colors [BlueOffset] * entry->usage ;
          itemcount += entry->usage ;
        }
      }
    }

    if (itemcount == 0)
      return ;

    color_map [color].red = (red + itemcount/2) / itemcount ;
    color_map [color].green = (green + itemcount/2) / itemcount ;
    color_map [color].blue = (blue + itemcount/2) / itemcount ;

    return ;
  }

  //
  // Description:
  //    This function finds the largest dimension of a color area.
  //
  // Parameters:
  //    area:    The color area to use
  //
  int BitmapImage::LargestColorRange (ColorArea &area)
  {
    unsigned int deltared = area.color_values [RedOffset].high
                          - area.color_values [RedOffset].low ;
    unsigned int deltagreen = area.color_values [GreenOffset].high
                          - area.color_values [GreenOffset].low ;
    unsigned int deltablue = area.color_values [BlueOffset].high
                          - area.color_values [BlueOffset].low ;

    if (deltared >= deltagreen && deltared >= deltablue)
      return RedOffset ;

    if (deltablue >= deltagreen && deltablue >= deltared)
      return BlueOffset ;

    return GreenOffset ;
  }

  //
  // Description:
  //    This function returns the 8-bit quantized color value for and RGB color.
  //
  // Parameters:
  //    red, green, blue: The RGB value to convert
  //
  unsigned int BitmapImage::QuantizedColor (UBYTE1 red, UBYTE1 green, UBYTE1 blue)
  {
    for (unsigned int color = 0 ; color < color_area_count ; ++ color)
    {
      if (red >= color_areas [color].color_values [RedOffset].low
          && red <= color_areas [color].color_values [RedOffset].high
          && green >= color_areas [color].color_values [GreenOffset].low
          && green <= color_areas [color].color_values [GreenOffset].high
          && blue >= color_areas [color].color_values [BlueOffset].low
          && blue <= color_areas [color].color_values [BlueOffset].high)
      {
        return color ;
      }
    }
    throw EGraphicsException ("INTERNAL ERROR - color not quantized") ;
```

```
  DUMMY_RETURN // MSVC++ is too stupid to realize we can't get here.
}

//
// Description:
//    This function converts the RGB color values in the source image
//    to 8-bit quantized color values.
//
// Parameters:
//    src:  The source image
//
void BitmapImage::QuantizeSourceImage (const BitmapImage &src)
{
  for (unsigned int rr = 0 ; rr < image_height ; ++ rr)
  {
    CallProgressFunction (rr * 100 / image_height, 2, 2) ;

    UBYTE1 *srcdata = &src.image_data [rr * src.row_width] ;
    UBYTE1 *dstdata = &image_data [rr * row_width] ;
    for (unsigned int cc = 0 ; cc < image_width ; ++ cc)
    {
      UBYTE1 red = srcdata [3 * cc + RedOffset] ;
      UBYTE1 green = srcdata [3 * cc + GreenOffset]  ;
      UBYTE1 blue = srcdata [3 * cc + BlueOffset] ;
      dstdata [cc] = QuantizedColor (red, green, blue) ;
    }
  }
  CallProgressFunction (100, 2, 2) ;
  return ;
}
```

```
#ifndef __BMP_H
#define __BMP_H
//
// Copyright (c) 1997,1998 Colosseum Builders, Inc.
// All rights reserved.
//
// Colosseum Builders, Inc. makes no warranty, expressed or implied
// with regards to this software. It is provided as is.
//
// See the README.TXT file that came with this software for information
// on redistribution or send E-mail to info@colosseumbuilders.com
//
// o The user assumes all risk for using this software. The authors of this
//   software shall be liable for no damages of any kind.
//
// o If the source code is distributed then this copyright notice must
//   remain unaltered and any modification must be noted.
//
// o If this code is shipped in binary format the accompanying documentation
//   should state that "this software is based, in part, on the work of
//   Colosseum Builders, Inc."
//

//
// BMP Library.
//
//  Title:   Windows Bitmap Definitions
//
//  Author:  John M. Miano  miano@colosseumbuilders.com
//

#include "grexcept.h"

class EBmpFileReadError : public EGraphicsException
{
public:
  EBmpFileReadError () : EGraphicsException ("Error reading BMP input file") {}
  EBmpFileReadError (const EBmpFileReadError &be) : EGraphicsException (be) {}
  EBmpFileReadError &operator=(const EBmpFileReadError &ge)
  {
    this->EGraphicsException::operator=(ge) ;
    return *this ;
  }
} ;

class EBmpNotABmpFile : public EGraphicsException
{
public:
  EBmpNotABmpFile () : EGraphicsException ("Not a Windows Bitmap stream") {}
  EBmpNotABmpFile (const EBmpNotABmpFile &be) : EGraphicsException (be) {}
  EBmpNotABmpFile &operator=(const EBmpNotABmpFile &ge)
  {
    this->EGraphicsException::operator=(ge) ;
    return *this ;
  }
} ;

class EBmpCorruptFile : public EGraphicsException
{
public:
  EBmpCorruptFile () : EGraphicsException ("Corrupt Windows Bitmap stream") {}
  EBmpCorruptFile (const EBmpCorruptFile &be) : EGraphicsException (be) {}
  EBmpCorruptFile &operator=(const EBmpCorruptFile &ge)
  {
    this->EGraphicsException::operator=(ge) ;
    return *this ;
  }
} ;

class EBmpNotSupported : public EGraphicsException
{
public:
  EBmpNotSupported () : EGraphicsException ("Unsupported Bitmap Format") {}
  EBmpNotSupported (const EBmpNotSupported &be) : EGraphicsException (be) {}
  EBmpNotSupported &operator=(const EBmpNotSupported &ge)
  {
```

```
    this->EGraphicsException::operator=(ge) ;
    return *this ;
  }
} ;

endif
```

```
//
// Copyright (c) 1997,1998 Colosseum Builders, Inc.
// All rights reserved.
//
// Colosseum Builders, Inc. makes no warranty, expressed or implied
// with regards to this software. It is provided as is.
//
// See the README.TXT file that came with this software for information
// on redistribution or send E-mail to info@colosseumbuilders.com
//
// o The user assumes all risk for using this software. The authors of this
//   software shall be liable for no damages of any kind.
//
// o If the source code is distributed then this copyright notice must
//   remain unaltered and any modification must be noted.
//
// o If this code is shipped in binary format the accompanying documentation
//   should state that "this software is based, in part, on the work of
//   Colosseum Builders, Inc."
//

//
// BMP Decoder Library.
//
// Title:   BmpDecoder Class Implementation
//
// Author: John M. Miano  miano@colosseumbuilders.com
//
//

#include <windows.h>

#include "bmpdecod.h"
#include "stdafx.h"
//
//  Description:
//
//    Class default constructor
//
BmpDecoder::BmpDecoder ()
{
  return ;
}

//
//  Description:
//
//    Class Copy Constructor
//
BmpDecoder::BmpDecoder (const BmpDecoder &source)
{
  Initialize () ;
  DoCopy (source) ;
  return ;
}

//
//  Description:
//
//    Class Destructor
//
BmpDecoder::~BmpDecoder ()
{
  return ;
}

//
//  Description:
//
//    Assignment operator.
//
//  Parameters:
//   source: The object to copy
//
BmpDecoder &BmpDecoder::operator=(const BmpDecoder &source)
```

```
{
  DoCopy (source) ;
  return *this ;
}

//
//  Description:
//
//    Common class initialization function for use by constructors.
//
void BmpDecoder::Initialize ()
{
  return ;
}

//
//  Description:
//
//    Common class copy function.
//
//  Parameters:
//    source:  The object to copy.
//
void BmpDecoder::DoCopy (const BmpDecoder &source)
{
  BitmapImageDecoder::DoCopy (source) ;
  return ;
}

//
//  Description:
//
//    This function reads an image from a Windows BMP stream.
//
//  Parameters:
//    strm: The input stream
//    image: The image to be read
//
void BmpDecoder::ReadImage (std::istream &strm, BitmapImage &image)

  bool os2format ;

  // We need this because MSVC++ does not follow standard scoping
  // rules in for statements.
  unsigned int ii ;

  unsigned int bytesread = 0 ;

  BITMAPFILEHEADER fileheader ;
  strm.read ((char *) &fileheader, sizeof (fileheader)) ;
  if (strm.gcount () != sizeof(fileheader))
    throw EBmpFileReadError () ;
  bytesread += sizeof (fileheader) ;

  const UBYTE2 signature = 'B' | ('M' << 8) ;
  if (fileheader.bfType != signature)
    throw EBmpNotABmpFile () ;

  // The header can come in one of two flavors.  They both
  // begin with a DWORD headersize.

  DWORD headersize ;
  strm.read ((char *) &headersize, sizeof (headersize)) ;

  unsigned long width ;
  unsigned long height ;
  unsigned int bitcount ;
  unsigned int compression ;
  if (headersize == sizeof (BITMAPCOREHEADER))
  {
    // OS/2 Format Header

    BITMAPCOREHEADER header ;
    header.bcSize = headersize ;
    strm.read ((char *) &header.bcWidth,
          sizeof (header) - sizeof (headersize)) ;
```

```
      bytesread += sizeof (header) ;

      width = header.bcWidth ;
      height = header.bcHeight ;
      bitcount = header.bcBitCount ;

      compression = BI_RGB ;

      os2format = true ;
    }
    else if (headersize >= sizeof (BITMAPINFOHEADER))
    {
      BITMAPINFOHEADER header ;
      header.biSize = headersize ;
      strm.read ((char *) &header.biWidth,
            sizeof (header) - sizeof (headersize)) ;
      bytesread += sizeof (header) ;
      compression = header.biCompression ;

      width = header.biWidth ;
      height = header.biHeight ;
      bitcount = header.biBitCount ;

      for (unsigned int ii = 0 ;
           ii < headersize - sizeof (BITMAPINFOHEADER) ;
           ++ ii)
      {
        ++ bytesread ;
        UBYTE1 data ;
        strm.read ((char *) &data, 1) ;
      }
      os2format = false ;
    }
    else
    {
      throw EBmpCorruptFile () ;
    }

    // Calculate the number of colors and make sure that the
    // compression method is compatible.
    unsigned int colorcount ;
    switch (bitcount)
    {
    case 1:
      if (compression != BI_RGB)
        throw EBmpNotSupported () ;
      colorcount = 1 << bitcount ;
      break ;
    case 4:
      if (compression != BI_RGB && compression != BI_RLE4)
        throw EBmpNotSupported () ;
      colorcount = 1 << bitcount ;
      break ;
    case 8:
      if (compression != BI_RGB &&  compression != BI_RLE8)
        throw EBmpNotSupported () ;
      colorcount = 1 << bitcount ;
      break ;
    case 24:
      if (compression != BI_RGB)
        throw EBmpNotSupported () ;
      colorcount = 0 ;
      break ;
    default:
      throw EBmpNotSupported () ;
    }

    // Allocate storage for the image.
    image.SetSize (colorcount, bitcount, width, height) ;

    // Read the color map.
    if (os2format)
    {
      for (ii = 0 ; ii < colorcount ; ++ ii)
      {
        RGBTRIPLE color ;
```

```
      strm.read ((char *) &color, sizeof (color)) ;
      image.ColorMap (ii).red = color.rgbtRed ;
      image.ColorMap (ii).blue = color.rgbtBlue ;
      image.ColorMap (ii).green = color.rgbtGreen ;

      bytesread += sizeof (color) ;
    }
  }
  else
  {
    for (ii = 0 ; ii < colorcount ; ++ ii)
    {
      RGBQUAD color ;
      strm.read ((char *) &color, sizeof (color)) ;
      image.ColorMap (ii).red = color.rgbRed ;
      image.ColorMap (ii).blue = color.rgbBlue ;
      image.ColorMap (ii).green = color.rgbGreen ;

      bytesread += sizeof (color) ;
    }
  }

  // It is poss ible to have a file where the image data does not
  // immediately follow the color map (or headers). If there is
  // padding we skip over it.
  if (bytesread > fileheader.bfOffBits)
    throw EBmpCorruptFile () ;
  for (ii = bytesread ; ii < fileheader.bfOffBits ; ++ ii)
  {
    UBYTE1 data ;
    strm.read ((char *) &data, 1) ;
  }

  // Read the image data.
  CallProgressFunction (0) ;
  if (bitcount != 24)
  {
    // In this block we handle images that use a color map.

    if (compression == BI_RGB)
    {
      // Simplest case -- No compression. We can just read the
      // raw data from the file.

      // Number of bits required for each pixel row.
      unsigned int bitwidth = bitcount * width ;
      // Number of bytes need to store each pixel row.
      unsigned int rowwidth = (bitwidth + 7)/8 ;
      // Number of bytes used to store each row in the BMP file.
      // This is is rowwidth rounded up to the nearest 4 bytes.
      unsigned int physicalrowsize = (rowwidth + 0x3) & ~0x3 ;
      // The number of pad bytes for each row in the BMP file.
      unsigned int padsize = physicalrowsize -  rowwidth ;

      // Read in each row.
      for (unsigned int ii = 0 ; ii < height ; ++ ii)
      {
        CallProgressFunction (ii * 100 / height) ;

        // The pixel rows are stored in reverse order.
        unsigned int index = (height - ii - 1) ;
        strm.read ((char *)&image [index][0], rowwidth) ;
        if (strm.gcount () != rowwidth)
          throw EBmpFileReadError () ;

        // Skip over the pad bytes.
        static char pad [4] ;
        strm.read (pad, padsize) ;
      }
    } //
    else if (compression == BI_RLE8)
    {
      // Handle the case of 8-bit Run-Length Encoding.

      unsigned int row = height - 1 ;  // Current row
      unsigned int col = 0 ;           // Current column
```

```
// The mechanism here is the same as for BI_RLE8 with two
// exceptions. Here we are dealing with 4-bit nibbles rather
// than whole bytes. This results in some extra work. In
// addition, the coding of runs includes two color values.

unsigned int row = height - 1 ;
unsigned int col = 0 ;
bool done = false ;
while (! strm.eof () && ! done)
{
  CallProgressFunction ((height - row - 1) * 100 / height) ;

  struct
  {
    UBYTE1 count ;
    UBYTE1 command ;
  } opcode ;

  strm.read ((char *) &opcode, sizeof (opcode)) ;
  if (opcode.count == 0)
  {
    switch (opcode.command)
    {
    case 0:     // Advance to next pixel row
      -- row ;
      col = 0 ;
      break ;
    case 1:     // Image complete
      done = true ;
      break ;
    case 2:   // Move to relative location the image.
      {
        UBYTE1 dx ;
        UBYTE1 dy ;
        strm.read ((char *) &dx, 1) ;
        strm.read ((char *) &dy, 1) ;
        col += dx ;
        row -= dy ;
      }
      break ;
    default:
      {
        UBYTE1 data ;
        UBYTE1 hi ;
        UBYTE1 lo ;
        if (row >= height || col + opcode.command > width)
            throw EBmpCorruptFile () ;
        for (unsigned int ii = 0 ; ii < opcode.command ; ++ ii)
        {
          if ((ii & 1) == 0)
          {
            strm.read ((char *) &data, 1) ;
            lo = data & 0xF ;
            hi = (data & 0xF0) >> 4 ;
          }
          if ((col & 1) == 0)
          {
            if ((ii & 1) == 0)
            {
              image [row][col/2] = hi << 4 ;
            }
            else
            {
              image [row][col/2] = lo << 4 ;
            }
          }
          else
          {
            if ((ii & 1) == 0)
            {
             image [row][col/2] |= hi  ;
            }
            else
            {
              image [row][col/2] |= lo ;
            }
```

```
            }
            ++ col ;
          }
          // If the number of bytes used in this instruction
          // is odd then there is a padding byte.
          switch (opcode.command & 0x3)
          {
          case 1: case 2:
            strm.read ((char *) &data, 1) ;
            break ;
          }
        }
        break ;
      }
    }
    else
    {
      // Process a run of the same color value pairs.
      UBYTE1 hi = opcode.command >> 4 ;
      UBYTE1 lo = opcode.command & 0xF ;
      if (row >= height || col + opcode.count > width)
        throw EBmpCorruptFile () ;

      for (unsigned int ii = 0 ; ii < opcode.count ; ++ ii)
      {
        if ((col & 1) == 0)
        {
          if ((ii & 1) == 0)
          {
            image [row][col/2] = hi << 4 ;
          }
          else
          {
            image [row][col/2] = lo << 4 ;
          }
        }
        else
        {
          if ((ii & 1) == 0)
          {
           image [row][col/2] |= hi  ;
          }
          else
          {
            image [row][col/2] |= lo ;
          }
        }
        ++ col ;
      }
    }
  }
  if (! done)
    throw EBmpCorruptFile () ;
 }
 else
 {
  // Invalid compression type
  throw EBmpCorruptFile () ;
 }
}
else
{
  // Read the data for a 24-bit image.

  // Number of bytes used to store each pixel row in the
  // the file. This value is rounded up to the nearest
  // multiple of four.
  unsigned int physicalrowsize = (3 * width + 0x3) & ~0x3 ;
  // Size of the padding for each row.
  unsigned int padsize = physicalrowsize -  3 * width ;

  for (unsigned int yy = 0 ; yy < height ; ++ yy)
  {
    CallProgressFunction (yy * 100 / height) ;

    unsigned int index = height - yy - 1 ;
```

```
        // Have to read the sample values separately because the colors
        // are in reverse order: BGR. If you are on Windows you could
        // red the whole thing a row at a time.
        for (unsigned int xx = 0 ; xx < 3 * width ; xx += 3)
        {
          strm.read ((char *)&image[index][xx + BitmapImage::BlueOffset], 1) ;
          strm.read ((char *)&image[index][xx + BitmapImage::GreenOffset], 1) ;
          strm.read ((char *)&image[index][xx + BitmapImage::RedOffset], 1) ;
        }
        static char pad [4] ;
        strm.read (pad, padsize) ;
      }
    }
    CallProgressFunction (100) ;
    return ;
  }

  //
  //  Description:
  //
  //    This function is used to call the progres function.
  //
  //  Parameters:
  //    percent:  The percent complete (0..100)
  //
  void BmpDecoder::CallProgressFunction (unsigned int percent)
  {
    if (progress_function == NULL)
      return ;
    bool cancel = false ;
    progress_function (*this, progress_data, 1, 1, percent, cancel) ;
    if (cancel)
      throw EGraphicsAbort () ;
    return ;
  }
```

```
#ifndef __BMPDECOD_H
#define __BMPDECOD_H
//
// Copyright (c) 1997,1998 Colosseum Builders, Inc.
// All rights reserved.
//
// Colosseum Builders, Inc. makes no warranty, expressed or implied
// with regards to this software. It is provided as is.
//
// See the README.TXT file that came with this software for information
// on redistribution or send E-mail to info@colosseumbuilders.com
//
// o The user assumes all risk for using this software. The authors of this
//   software shall be liable for no damages of any kind.
//
// o If the source code is distributed then this copyright notice must
//   remain unaltered and any modification must be noted.
//
// o If this code is shipped in binary format the accompanying documentation
//   should state that "this software is based, in part, on the work of
//   Colosseum Builders, Inc."
//

//
//  BMP Decoder Library.
//
//  Title:   BmpDecoder Class Implementation
//
//  Author:  John M. Miano  miano@colosseumbuilders.com
//
//  Description:
//
//    This class decodes Windows BMP file.
//
//

#include <iostream>

#include "bitimage.h"
#include "grexcept.h"
#include "bmp.h"

class BmpDecoder : public BitmapImageDecoder
{
public:
  BmpDecoder () ;
  virtual ~BmpDecoder () ;
  BmpDecoder (const BmpDecoder &) ;
  BmpDecoder &operator=(const BmpDecoder &) ;

  virtual void ReadImage (std::istream &, BitmapImage &) ;
private:
  void Initialize () ;
  void DoCopy (const BmpDecoder &) ;
  void CallProgressFunction (unsigned int percent) ;
} ;

#endif
```

```
//
// Copyright (c) 1997,1998 Colosseum Builders, Inc.
// All rights reserved.
//
// Colosseum Builders, Inc. makes no warranty, expressed or implied
// with regards to this software. It is provided as is.
//
// See the README.TXT file that came with this software for information
// on redistribution or send E-mail to info@colosseumbuilders.com
//
// o The user assumes all risk for using this software. The authors of this
//   software shall be liable for no damages of any kind.
//
// o If the source code is distributed then this copyright notice must
//   remain unaltered and any modification must be noted.
//
// o If this code is shipped in binary format the accompanying documentation
//   should state that "this software is based, in part, on the work of
//   Colosseum Builders, Inc."
//

//
// Title:  Windows Bitmap Coder
//
// Author: John M. Miano miano@colosseumbuilders.com
//

#include <windows.h>
#include <iostream.h>
#include "bmpencod.h"
#include "stdafx.h"

const UBYTE2 Signature = 'B'|('M'<<8) ;

//
//  Description:
//
//    Default class constructor
//
BmpEncoder::BmpEncoder ()
{
  Initialize () ;
  return ;
}

//
//  Description:
//
//    Class copy constructor
//
BmpEncoder::BmpEncoder (const BmpEncoder &source)
{
  Initialize () ;
  DoCopy (source) ;
  return ;
}

//
//  Descriptor:
//
//    Class destructor
//
BmpEncoder::~BmpEncoder ()
{
  return ;
}

//
//  Description:
//
//    Class assignment operator.
//
//  Parameters:
//    source:  The object to copy
//
BmpEncoder &BmpEncoder::operator=(const BmpEncoder &source)
```

```
{
  DoCopy (source) ;
  return *this ;
}

//
//  Description:
//
//    Common object initialization
//
void BmpEncoder::Initialize ()
{
  // For now, a NoOp.
  return ;
}

//
//  Description:
//
//    Common object copy function.
//
//  Parameters:
//    source:  The object to copy
//
void BmpEncoder::DoCopy (const BmpEncoder &source)
{
  BitmapImageEncoder::DoCopy (source) ;
  return ;
}

//
//  Description:
//
//    This function writes an image to a BMP stream.
//
//  Parameters:
//    strm:  The output stream
//    image:  The image to output
//
void BmpEncoder::WriteImage (std::ostream &strm, BitmapImage &image)
{
  // We need this because MSVC++ does not follow standard scoping rules
  // in for statements
  unsigned int ii ;

  switch (image.BitCount ())
  {
  case 1: case 4: case 8: case 24:
    break ;
  default:
    throw EBmpNotSupported () ;
  }

  BITMAPFILEHEADER fileheader = {
          SystemToLittleEndian (Signature),
          0,
          0,
          0,
          0 } ;
  BITMAPINFOHEADER infoheader = {
          SystemToLittleEndian ((UBYTE4)sizeof (BITMAPINFOHEADER)),
          SystemToLittleEndian ((UBYTE4)image.Width ()),
          SystemToLittleEndian ((UBYTE4)image.Height ()),
          SystemToLittleEndian ((UBYTE2) 1), // Planes
          0,                  // biBitCount
          SystemToLittleEndian ((UBYTE4) BI_RGB), // biCompression
          0,                  // biXPelsPerMeter
          0,                  // biYPelsPerMeter
          0,                  // biClrUsed
          0,                  // biClrImportant
          } ;
  unsigned int colorsize = sizeof (RGBQUAD)
                           * image.ColorCount () ;

  // Determine the amount of space required to store each
  // row of the image.
```

```
unsigned int outputwidth ;
if (image.BitCount () != 24)
{
  unsigned int bitwidth = image.BitCount () * image.Width () ;
  outputwidth = (bitwidth + 7)/8 ;
}
else
{
  outputwidth = sizeof (RGBTRIPLE) * image.Width () ;
}

// Find the amount of space required to pad the output rows to
// a multiple of four bytes.
unsigned int padsize = ((outputwidth + 0x3) & ~0x3) - outputwidth ;

// Calulate the space required for the image.
unsigned int datasize = image.Height () * (outputwidth + padsize) ;
unsigned int spacerequired = sizeof (BITMAPFILEHEADER)
                           + sizeof (BITMAPINFOHEADER)
                           + colorsize + datasize ;

// Fill in the remaining header fields.
fileheader.bfOffBits = SystemToLittleEndian ((UBYTE4)sizeof (BITMAPFILEHEADER)
                     + sizeof (BITMAPINFOHEADER)
                     + colorsize) ;
fileheader.bfSize = SystemToLittleEndian ((UBYTE4) spacerequired) ;
infoheader.biBitCount = SystemToLittleEndian ((UBYTE2) image.BitCount ()) ;
// Write the header.
strm.write ((char *) &fileheader, sizeof (BITMAPFILEHEADER)) ;
strm.write ((char *) &infoheader, sizeof (BITMAPINFOHEADER)) ;

for (ii = 0 ; ii < image.ColorCount () ; ++ ii)
{
  RGBQUAD data ;
  data.rgbRed = image.ColorMap (ii).red ;
  data.rgbGreen = image.ColorMap (ii).green ;
  data.rgbBlue = image.ColorMap (ii).blue ;
  data.rgbReserved = 0 ;
  strm.write ((char *) &data, sizeof (RGBQUAD)) ;
}

CallProgressFunction (0) ;
if (image.BitCount () != 24)
{
  for (ii = 0 ; ii < image.Height () ; ++ ii)
  {
    CallProgressFunction (ii * 100 /image.Height ()) ;
    static const char pad [4] = { 0, 0, 0, 0, } ;
    unsigned int index = image.Height () - ii - 1 ;
    strm.write ((char* ) &image [index][0], outputwidth) ;
    strm.write (pad, padsize) ;
  }
}
else
{
  for (ii = 0 ; ii < image.Height () ; ++ ii)
  {
    CallProgressFunction (ii * 100 /image.Height ()) ;
    unsigned int index = image.Height () - ii - 1 ;
    for (unsigned int ii = 0 ; ii < 3 * image.Width () ; ii += 3)
    {
      // Remember BMP puts the colors in reverse order BGR.
      strm.write ((char *) &image [index][ii+BitmapImage::BlueOffset], 1) ;
      strm.write ((char *) &image [index][ii+BitmapImage::GreenOffset], 1) ;
      strm.write ((char *) &image [index][ii+BitmapImage::RedOffset], 1) ;
    }
    static const char pad [4] = { 0, 0, 0, 0, } ;
    strm.write (pad, padsize) ;
  }
}
CallProgressFunction (100) ;
return ;


//
//  Description:
```

```
//
//    This function calls the progress function if it has been defined.
//
//  Parameters:
//    percent: The percent completed (0..100)
//
void BmpEncoder::CallProgressFunction (unsigned int percent)
{
  if (progress_function == NULL)
    return ;
  bool cancel = false ;
  progress_function (*this, progress_data, 1, 1, percent, cancel) ;
  if (cancel)
    throw EGraphicsAbort () ;
  return ;
 }
```

```
#ifndef __BMBENCOD_H
#define __BMPENCOD_H

//
// Copyright (c) 1997,1998 Colosseum Builders, Inc.
// All rights reserved.
//
// Colosseum Builders, Inc. makes no warranty, expressed or implied
// with regards to this software. It is provided as is.
//
// See the README.TXT file that came with this software for information
// on redistribution or send E-mail to info@colosseumbuilders.com
//
// o The user assumes all risk for using this software. The authors of this
//   software shall be liable for no damages of any kind.
//
// o If the source code is distributed then this copyright notice must
//   remain unaltered and any modification must be noted.
//
// o If this code is shipped in binary format the accompanying documentation
//   should state that "this software is based, in part, on the work of
//   Colosseum Builders, Inc."
//

//
//  Title:  Windows Bitmap Encoder Class
//
//  Author:  John M. Miano  miano@colosseumbuilders.com
//

#include <iostream>

#include "bitimage.h"
#include "bmp.h"

class BmpEncoder : public BitmapImageEncoder
{
public:
  BmpEncoder () ;
  BmpEncoder (const BmpEncoder &) ;
  virtual ~BmpEncoder () ;
  BmpEncoder &operator=(const BmpEncoder &) ;
  virtual void WriteImage (std::ostream &, BitmapImage &) ;
private:
  void Initialize () ;
  void DoCopy (const BmpEncoder &) ;
  void CallProgressFunction (unsigned int) ;
} ;

#endif
```

```
// ChildFrm.cpp : implementation of the CChildFrame class
//

#include "stdafx.h"
#include "EncryptThis.h"

#include "ChildFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CChildFrame

IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)

BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
    //{{AFX_MSG_MAP(CChildFrame)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CChildFrame construction/destruction

CChildFrame::CChildFrame()
{
    // TODO: add member initialization code here

}

CChildFrame::~CChildFrame()
{
}

BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
    //  the CREATESTRUCT cs

    return CMDIChildWnd::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CChildFrame diagnostics

#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
    CMDIChildWnd::AssertValid();
}

void CChildFrame::Dump(CDumpContext& dc) const
{
    CMDIChildWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CChildFrame message handlers
```

```
// ChildFrm.h : interface of the CChildFrame class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_CHILDFRM_H__6675968C_A327_11D2_B20A_444553540000__INCLUDED_)
#define AFX_CHILDFRM_H__6675968C_A327_11D2_B20A_444553540000__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CChildFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE(CChildFrame)
public:
    CChildFrame();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CChildFrame)
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CChildFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
    //{{AFX_MSG(CChildFrame)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the previous
line.

#endif // !defined(AFX_CHILDFRM_H__6675968C_A327_11D2_B20A_444553540000__INCLUDED_)
```

```
// EncryptAlgorithm.cpp: implementation of the EncryptAlgorithm class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "EncryptThis.h"
#include "EncryptAlgorithm.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

EncryptAlgorithm::EncryptAlgorithm()
{
}

EncryptAlgorithm::EncryptAlgorithm(CBitmap *img, CView *cwin)
{
}

EncryptAlgorithm::~EncryptAlgorithm()
{

}
```

```
// EncryptAlgorithm.h: interface for the EncryptAlgorithm class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_ENCRYPTALGORITHM_H__74F57EA0_A32C_11D2_B20A_444553540000__INCLUDED_)
#define AFX_ENCRYPTALGORITHM_H__74F57EA0_A32C_11D2_B20A_444553540000__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
#include "stdafx.h"

class EncryptAlgorithm
{
public:
    EncryptAlgorithm();
    EncryptAlgorithm(CBitmap *img, CView *cwin);
    virtual ~EncryptAlgorithm();

    virtual void GetOptions() = 0;  // GetOptions loads up the dialog box for changing the encrypt
ion options
    virtual void EncryptThis(BitmapImage* bmpImage) = 0;
    virtual void Disable() = 0;
    virtual void Enable() = 0;
    virtual CBitmap* GetDisplayImage() = 0;
    virtual void SaveEncData(CString file) = 0;
};

#endif // !defined(AFX_ENCRYPTALGORITHM_H__74F57EA0_A32C_11D2_B20A_444553540000__INCLUDED_)
```